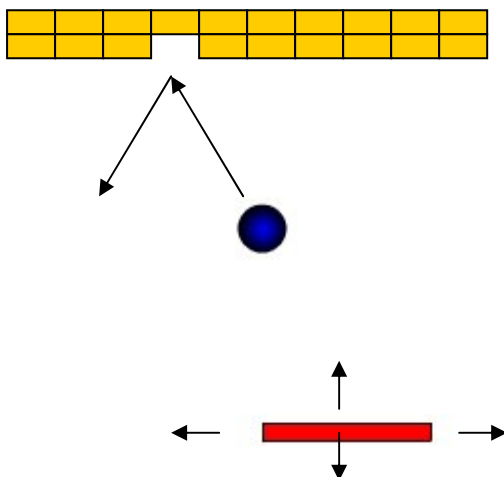


Games Scripting and Programming Workshop

Using action scripting as a basis for Object Oriented Programming

Flash Actionscript 2.0



Allan Morrison
Senior Project Officer
Museum Magnet Schools
Queensland Museum
P.O. Box 3300, South Bank, Qld. 4101,
Australia.
Web: <http://www.mms.qld.edu.au>
Email: allan.morrison@qm.qld.gov.au or
mms@qm.qld.gov.au
Ph. 07 3840 7611
Fax 07 3840 7607



Introduction to Object Oriented Programming (OOP)

Flash Actionscript 2.0

Overview

Why is it necessary to learn about OOP in flash? What is wrong with my old way of doing things? The answer to both of these questions lies with the answer to yet another question:

What do you want to use flash for?

If the answer is to develop complex games that involve the interactions of large numbers of game objects independent of a prescribed timeline, then OOP may be for you. For relatively simple games that involve few game objects and interactions are based on progression through a timeline, then traditional, "procedural" methods of programming in flash are more than adequate. These methods involve embedding actionscript into either frames on the timeline that determine outcomes based on game status or objects that have embedded code that responds to particular events.

The issues that arise with this kind of programming include:

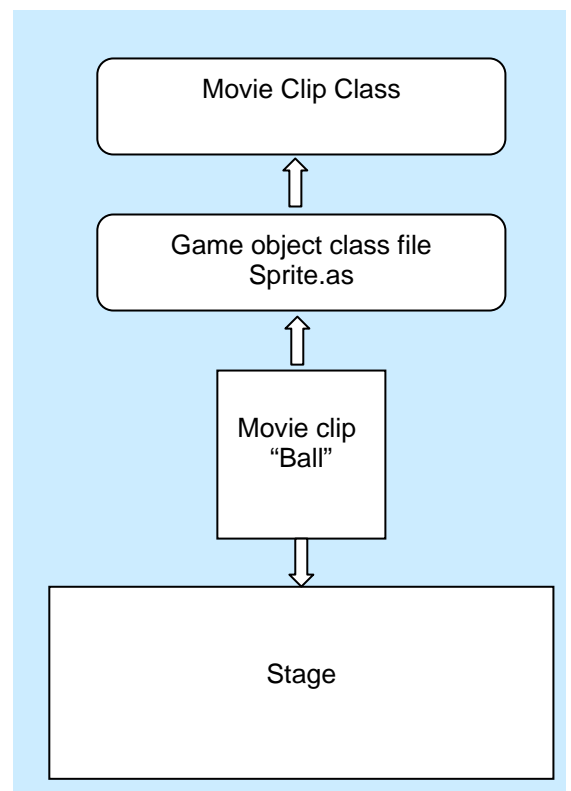
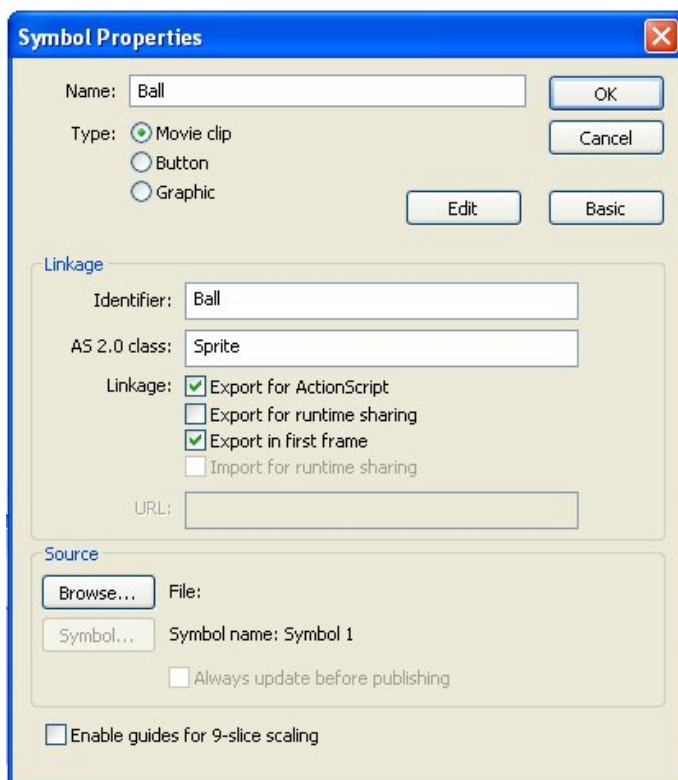
- Code tends to be difficult to manage as it is located in a variety of locations such as frames and objects
- Code tends to be inefficient due to repetition of code among similar objects.
- Debugging and error checking can be problematical due to the embedding process that makes it difficult to locate the offending code.

Clearly, there must be a better way. A solution can be found using OOP.

Try <http://www.totebo.com/smashing.php>

Getting started

Essentially, an object contains its properties and methods in a separate class or actionscript file. The file.as can be written in any editor and saved as a text file. The file is linked to the associated object through its linkage properties. Objects are created as movieclips thereby inheriting inbuilt [movieclip properties and methods](#). Setting the linkage properties as shown, creates the following associations:



What follows is a more detailed look at the approach to OOP in flash actionscript.

What does the basic “Sprite.as” class file look like?

```
class Sprite extends MovieClip {
    //
    //===== Declare variables=====
    //

        // variables in class file

    //
    //===== Constructor =====
    //
    //Constructor sets the initial properties of square

    public function Sprite() {

        // set values here

    }
    //
    //===== Methods =====
    //

        // how to do stuff here

    //
    //===== Getters =====
    //

        // getters return a value of sprite properties

    //
    //===== Setters =====
    //

        // setters pass a new value for properties of sprite

}
}
```

Note that class declaration name (Sprite) has the same name as the constructor function (Sprite) as well as the class file name (Sprite) including capitalisation.

All class files have similar structure:

1. Class file declaration
2. Declare variables
3. Constructor
4. Methods
5. Getters and setters



Development of an OOP arcade game style project “Brick”

Phase 1

Add a movie clip to the screen that moves within defined boundaries

1. Create a class file for a sprite that defines its properties and methods.
2. Create a flash game file and a game object
3. Set linkage between the game object and the class file
4. Add actionscript to the flash game file to initiate the action

1. Class file for a sprite

```
class Sprite extends MovieClip {
//
//===== Declare variables=====
//
    var spX :Number;
    var spY :Number;
//
//===== Constructor =====
//
    public function Sprite() {
        // set starting speed
        spX = 10;
        spY = 10;
        // set starting position
        this._x = Stage.width/2;
        this._y = Stage.height/3*2;
    }
//
//===== Methods =====
//
// move Sprite method
public function moveSprite():Void {
    //move forward at set speed
    this._x += spX;
    this._y += spY;
    //check if top or bottom boundary reached
    if ((this._y>=Stage.height-10) or (this._y<=5)){
        //rebound y
        reBoundY();
    }
    //check if right or left boundary reached
    if ((this._x>=Stage.width-10) or (this._x<=5)){
        //rebound x
        reBoundX();
    }
}
// rebound x direction
public function reBoundX():Void {
    // change x direction
    spX = -spX;
    // give boost
    this._x += 2*spX;
}
}
```

```

// rebound y direction
public function reBoundY():Void {
    //change y direction
    spY = -spY;
    // give boost
    this._y += 2*spY;
}

//
// ===== Getters =====
//
//get x position of sprite
public function getX():Number {
    //gets current position and returns value
    return this._x;
}
//get y position of sprite
public function getY():Number {
    //gets current position and returns value
    return this._y;
}
//get width of sprite
public function getWidth():Number {
    //gets width of sprite and returns the number
    return this._width;
}
//get height of sprite
public function getHeight():Number {
    //gets height of sprite and returns the number
    return this._height;
}
//
// ===== Setters =====
//
//set initial x position of sprite
public function setX(_xpos:Number):Void {
    var xpos:Number = _xpos;
    //sets current position to value passed in parameter _xpos
    this._x = xpos;
}
//set Y position of sprite
public function setY(_ypos:Number):Void {
    var ypos:Number = _ypos;
    //sets current position to value passed in parameter _ypos
    this._y = ypos;
}
//set width of sprite
public function setWidth(_wd:Number) :Void {
    var wd:Number = _wd;
    //sets new width
    this._width = wd;
}
//set height of sprite
public function setHeight(_ht:Number) :Void {
    var ht:Number = _ht;
    //sets new width
    this._height = ht;
}
}

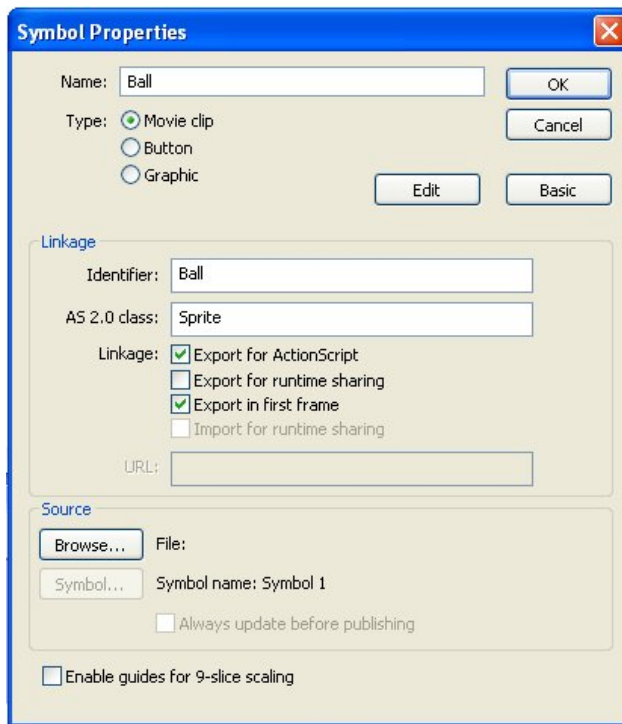
```



2. Example Flash file

- Start a new flash file and save it as Game.fla.
- Create a ball object and convert to the symbol Ball saved in the library.
- Remove the ball from the stage.

3. Set the properties of the Ball movie clip as follows:



4. In frame 1 of layer 1 of the flash file, add the following actionscript:

- Add ball to stage
- Move the ball each time the movie loops

```
// attach movie to stage
attachMovie("Ball", "ball", 100);

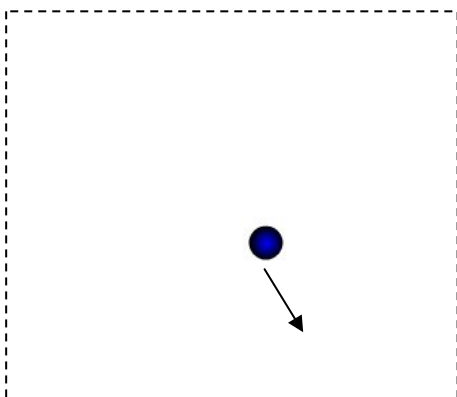
// loop
this.onEnterFrame = function() {
    ball.moveSprite();
}
```

This script creates a new instance of the Ball movieclip and places it on the stage in layer 100 according to the initial values in the constructor.

Each time the file loops back to the frame it executes the onEnterFrame function that calls the moveSprite method of the Sprite class.



Checkpoint 1



Functionality achieved:

- Ball appears on stage in a given starting position
- Ball moves across the stage at a set speed
- Ball rebounds off defined boundaries

Next phase of development:

- add a sprite that can be controlled



Phase 2

Add a sprite that can be controlled

Control of on screen objects is achieved through the use of listeners. Listeners respond to keystrokes or mouse action events. The listener is placed in the actionscript of the first frame of the flash file as this is an interface layer event.

1. Create an object to be controlled

- Create a bat movie clip in the library.
- Set the properties of the bat movie clip as follows:

The screenshot shows the 'Symbol Properties' dialog box. The 'Name' field contains 'Bat'. The 'Type' is set to 'Movie clip'. The 'Linkage' section is expanded, showing 'Identifier' as 'Bat', 'AS 2.0 class' as 'Sprite', and three checked options: 'Export for ActionScript', 'Export in first frame', and 'Import for runtime sharing'. The 'Source' section shows 'File' and 'Symbol name: Symbol 1'. There are 'OK', 'Cancel', 'Edit', and 'Basic' buttons.

2. In frame 1 of layer 1 of the flash file, add the bat movie to the actionscript:

- Add bat to stage

```
// attach movie to stage
attachMovie("Ball", "ball", 100);
attachMovie("Bat", "bat", 200);

// add mouse listener here

// add keyboard listener

// loop
this.onEnterFrame = function() {
    ball.moveSprite();
}
```



Test your project now



3. Keyboard listener

A keyboard listener can be used to respond to the arrow keys for bat movement control. Place the following code in the action script panel for the first frame of the game file:

- Add the keyboard listener

```
// add keyboard listener
// instantiate new keyboard listener object
var keyListener:Object = new Object();
var dir:String;

keyListener.onKeyDown = function() {
    if (Key.isDown(Key.RIGHT)) {
        dir = "right";
    } else if (Key.isDown(Key.LEFT)) {
        dir = "left";
    } else if (Key.isDown(Key.UP)) {
        dir = "up";
    } else if (Key.isDown(Key.DOWN)) {
        dir = "down";
    }
}
keyListener.onKeyUp = function() {
    dir = "stop";
}

// add key listener method to key properties
Key.addListener(keyListener);
```

The listener will determine the direction arrow key pressed. The direction "dir" now needs to be sent to the sprite method that moves the bat in the appropriate direction. This is done in the onEnterFrame function as it needs to be moved each time the game loops.

- Add call to move function in game loop for the bat

```
// loop
this.onEnterFrame = function() {
    ball.moveSprite();
    bat.moveDir(dir);
}
```

4. In the sprite class file (Sprite.as) add the following method:

- Add the movement function to the sprite class

```
public function moveDir(_dir:String):Void {
    var dir:String = _dir;
    if(dir=="left")
        this._x -= spX;
    if(dir=="right")
        this._x += spX;
    if(dir=="up")
        this._y -= spY;
    if(dir=="down")
        this._y += spY;
    if(dir=="stop"){
        this._x +=0;
        this._y +=0;
    }
}
```



Test your project now



5. Mouse listener

- Add the mouse listener code to the first frame action script

```
// add mouse listener here
// instantiate new mouse listener object
var mouseListener:Object = new Object();
var isDragging:Boolean = false;

// define mouse drag event
mouseListener.onMouseDown = function() {
    this.isDragging = true;
}
mouseListener.onMouseMove = function() {
    if(this.isDragging) {
        bat.setX(_xmouse-bat.getWidth()/2);
        bat.setY(_ymouse-bat.getHeight()/2);
    }
}
mouseListener.onMouseUp = function() {
    this.isDragging = false;
}
// add mouse listener method to mouse properties
Mouse.addListener(mouseListener);

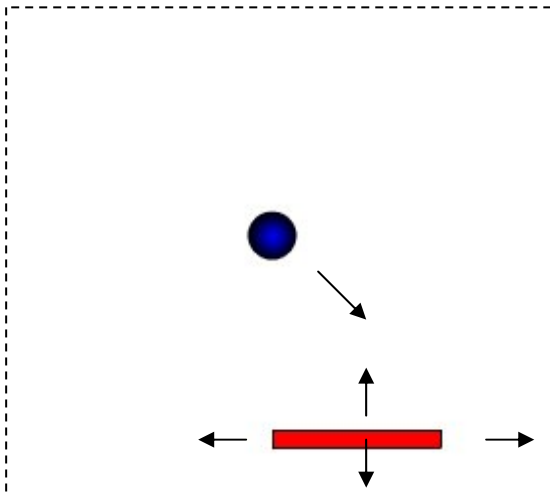
// add keyboard listener
```

Please note that startDrag and stopDrag methods only apply to movie clips and not to timeline events. Hence these methods are not available in the frame timeline actionscript unless they refer to a movie clip instance directly. There is a way of adding the startDrag and stopDrag methods to the movie clip methods in the class file. This provides the more eloquent solution. This can be investigated at a later time.



Checkpoint 2

Test your project now



Functionality achieved:

- All phase one functions
- Bat can be controlled by either a mouse drag or keyboard direction keys.

Next phase of development:

- add an interaction between bat and ball sprites

Notes

Summary of actionscript code on timeline:

```
// attach movie to stage
attachMovie("Ball", "ball", 100);
attachMovie("Bat", "bat", 200);

// add mouse listener
// instantiate new mouse listener object
var mouseListener:Object = new Object();
var isDragging:Boolean = false;

// define mouse drag event
mouseListener.onMouseDown = function() {
    this.isDragging = true;
}
mouseListener.onMouseMove = function() {
    if(this.isDragging) {
        //set bat centre to mouse cursor position
        bat.setX(_xmouse-bat.getWidth()/2);
        bat.setY(_ymouse-bat.getHeight()/2);
    }
}
mouseListener.onMouseUp = function() {
    this.isDragging = false;
}
// add mouse listener method to mouse properties
Mouse.addListener(mouseListener);

// add keyboard listener
// instantiate new keyboard listener object
var keyListener:Object = new Object();
var dir:String;

// define key down event
keyListener.onKeyDown = function() {
    if (Key.isDown(Key.RIGHT)) {
        dir = "right";
    } else if (Key.isDown(Key.LEFT)) {
        dir = "left";
    } else if (Key.isDown(Key.UP)) {
        dir = "up";
    } else if (Key.isDown(Key.DOWN)) {
        dir = "down";
    }
}

//define key up event
keyListener.onKeyUp = function() {
    dir = "stop";
}

// add key listener method to key properties
Key.addListener(keyListener);

// loop
this.onEnterFrame = function() {
    //trace(dir);
    ball.moveSprite();
    bat.moveDir(dir);
}
```

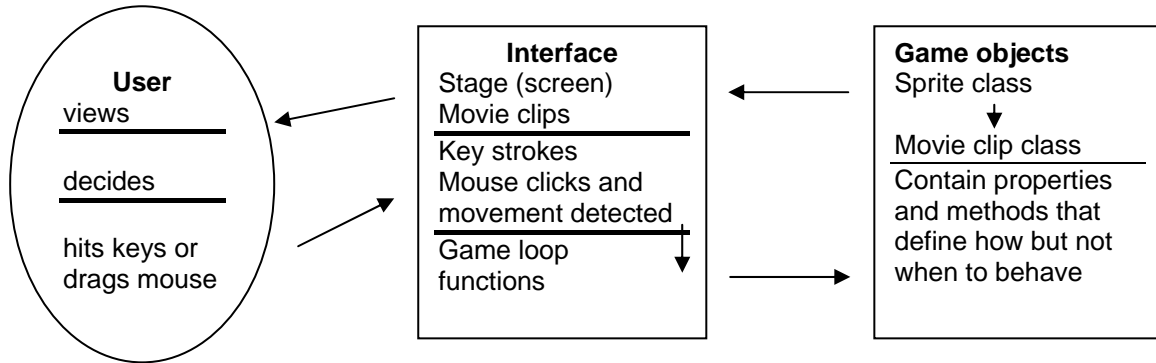


Phase 3

Add an interaction between bat and ball sprites

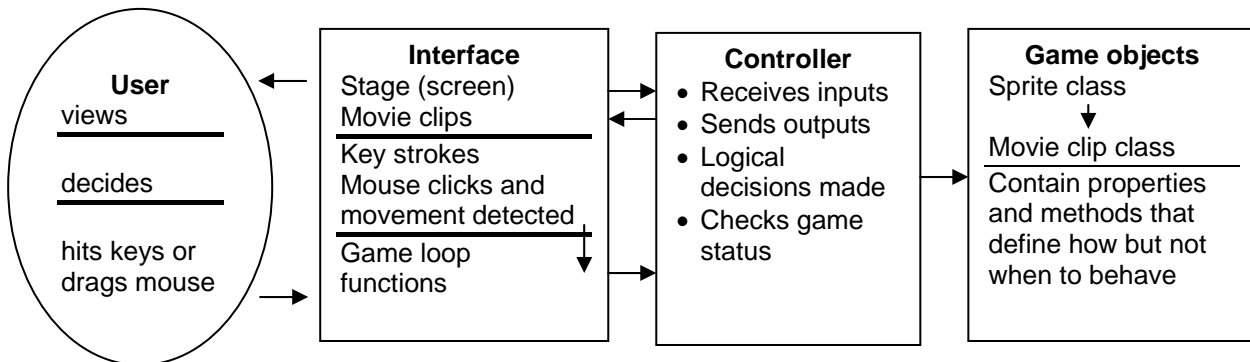
What coding structure has been achieved so far and what is the plan for the future? Some adjustment to the structure will be required to achieve a more defined and logical framework for all code to be organised under. This makes coding easier and more robust.

1. Interaction pathways according to current coding structure:



This model is limited due to the lack of provision for decision making to occur beyond determination of key strokes or mouse movement. There is no logical place to put code that deals with interactions of game objects. Hence a new code layer is required: the logic or control layer to deal with all the game logic including interactions, scoring, task completion, etc.

The control layer is placed between the Interface and Game Object layers in a kind of gatekeeper role. The control layer receives and sends messages to the interface from the game object layer. The control layer tells the game object *when* to invoke methods and read and change properties. A suitable model for organising code might be structured as follows:



2. Create a new actionscript class file called Game.as in the editor. Note the name of the file, the class name, and the constructor function must all have the same name with capitals: Game.

The constructor initialises the game objects ball and bat as part of the game. The game is instantiated in the interface layer in the frame action script.

A new method has been created that will control all sprite movement: moveSprites(). This public method is called from the game loop.



2.1. Class file: Game.as

- Add game class code to a new actionscript file saved as Game.as

```
class Game {
    //===== declare variables
    private var ball:Sprite;
    private var bat:Sprite;
    private var dir:String;

    //===== constructor
    public function Game (_ball:Sprite, _bat:Sprite) {
        ball=_ball;
        bat=_bat;
    }
    //===== public methods
    // move all sprites
    public function moveSprites(_dir):Void{
        dir = _dir;
        moveBall();
        moveBat(dir);
    }
    // ===== private methods
    //move ball
    private function moveBall() :Void {
        ball.moveSprite();
    }
    //move bat
    private function moveBat(_dir:String):Void {
        bat.moveDir(_dir);
    }
}
```

2.2. Frame action script

- Add new game class
- Modify code for game class methods

```
// attach movie to stage
attachMovie("Ball", "ball", 100);
attachMovie("Bat", "bat", 200);

// instantiate new game
//setup
var myGame:Game = new Game(ball, bat);

// loop
this.onEnterFrame = function() {
    myGame.moveSprites(dir);
    //ball.moveSprite();
    //bat.moveDir(dir);
}
```

2.3. Sprite class

No changes or additions are required here because the same methods are now called from the controller layer (Game.as) rather directly from the interface layer (frame actionscript).



Checkpoint 3

Test your project now

Test your new code to to make sure that all functionality of Checkpoint 2 is still maintained.



Phase 4

Add collision interaction between the bat and ball.

A collision test needs to be added to the game class. In this case, two methods are added: a public method called from the game loop and a private method that performs a hit test between any two movie sprites. The changes to the Game.as class file and the frame action script are shown below:

1. Class file: Game.as

- Add collision test functions

```
class Game {
    //===== declare variables
    private var ball:Sprite;
    private var bat:Sprite;
    private var dir:String;

    //===== constructor
    public function Game (_ball:Sprite, _bat:Sprite) {
        ball=_ball;
        bat=_bat;
    }
    //===== public methods
    // move all sprites
    public function moveSprites(_dir):Void{
        dir = _dir;
        moveBall();
        moveBat(dir);
    }
    //check collisions
    public function checkCollisions():Void {
        if(isCollision(bat,ball)) {
            // do something
            ball.reBoundY();
        }
    }
    // ===== private methods
    // move the ball
    private function moveBall() :Void {
        ball.moveSprite();
    }
    // move the bat in direction
    private function moveBat(_dir:String):Void {
        bat.moveDir(_dir);
    }
    // hit test two movie clips
    private function isCollision(_mc1:Sprite,_mc2:Sprite):Boolean {
        if(_mc1.hitTest(_mc2))
            return true;
    }
}
```

2. Frame action script

- Call game check collision function

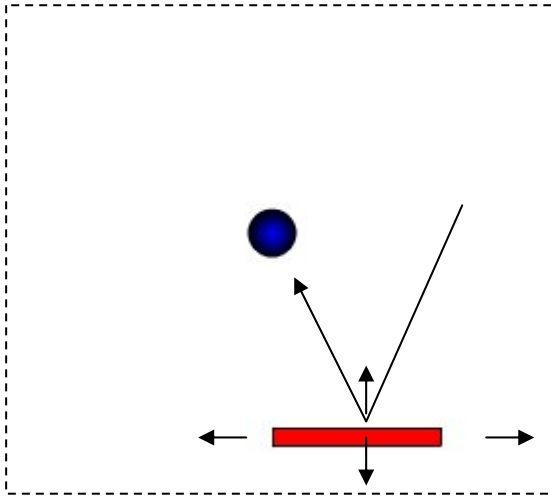
```
// loop
this.onEnterFrame = function() {
    myGame.moveSprites(dir);
    myGame.checkCollisions();
}
```





Checkpoint 4

Test your project now



Functionality achieved:

- All phase two functions
- Ball bounces off bat

Next phase of development:

- add an interaction between the ball and a brick sprite

Notes:



Phase 5

Add an interaction between ball and a brick sprite

- Create another game object, Brick, as a movieclip symbol with the same linkage to the Sprite class.
- Make sure all symbol properties are correctly set as before. Link to Sprite.as class.

The brick will have all the properties and methods of the sprite class even though not all will be used. For example, movement will not be required. An additional method is required to enable the brick to be removed from the screen. If added to the Sprite class, all sprites will gain this same method to be removed. This is not a bad thing as it may be useful to remove any sprite in the future. The interaction required is summarised as follows:

A ball rebounds off the brick and the brick disappears so as not take any further part in the game.

The additional code for the Sprite class, frame action script, and Game class follows:

1. Sprite class methods

- Add the remove function

```
public function reBoundX():Void {
    spX=-spX;
    this._x+= 2*spX;
}
public function reBoundY():Void {
    spY=-spY;
    this._y+= 2*spY;
}
public function reMove():Void {
    var mTemp:MovieClip = this.getInstanceAtDepth(0);
    this.swapDepths(0);
    this.removeMovieClip();
    if(mTemp != undefined) {
        mTemp.swapDepths(0);
    }
}
```

2. Frame action script

- Place the brick on the stage
- Add brick to game

```
// attach movie to stage
attachMovie("Ball", "ball", 100);
attachMovie("Bat", "bat", 200);
attachMovie("Brick", "brick", 300);
//position brick on stage at top left
brick._x=5;
brick._y=10;
// instantiate new game
var myGame:Game = new Game(ball, bat, brick);
```



3. Game controller class

- Add the brick to the game constructor
- Add the ball and brick collision test

```
class Game {
    //===== declare variables
    private var dir:String;
    private var ball:Sprite;
    private var bat:Sprite;
    private var brick:Sprite;

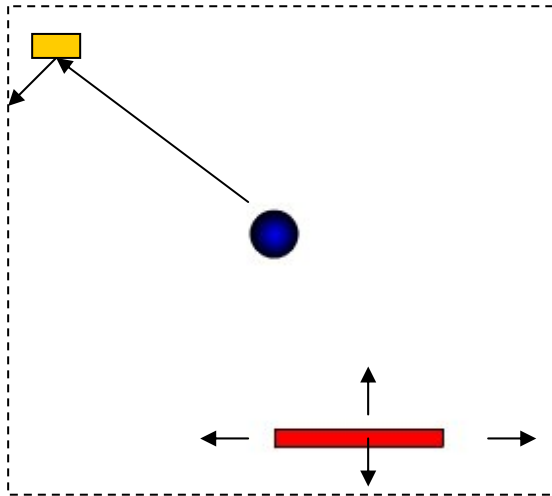
    //===== constructor
    public function Game (_ball:Sprite, _bat:Sprite, _brick:Sprite) {
        ball=_ball;
        bat=_bat;
        brick=_brick;
    }
    //===== public methods
    // move all sprites
    public function moveSprites(_dir:String):Void {
        dir = _dir;
        moveBall();
        moveBat(dir);
    }
    }
    public function checkCollisions():Void {
        if(isCollision(bat,ball)) {
            // do something
            ball.reBoundY();
        }
        if(isCollision(brick,ball)) {
            // do something
            ball.reBoundY();
            brick.reMove();
        }
    }
    // ===== private methods
    // move the ball
    private function moveBall() :Void {
        ball.moveSprite();
    }
    // move the bat in direction
    private function moveBat(_dir:String):Void {
        bat.moveDir(_dir);
    }
    // hit test two movie clips
    private function isCollision(_mc1:Sprite,_mc2:Sprite):Boolean {
        if(_mc1.hitTest(_mc2))
            return true;
    }
}
```





Checkpoint 5

Test your project now



Functionality achieved:

- All phase four functions
- Ball bounces off brick which is then removed

Next phase of development:

- add more bricks

Notes:



Phase 6

Add more bricks

This development phase highlights a key advantage of the OOP approach to coding. Introducing a large number of sprites is a relatively simple task to create and manage. The array dynamic data structure is used as a container to store and manage the brick sprites. Creation and management is achieved through simple for ... loops that cycle the same tasks through each brick in the array. Importantly, all the properties and methods of the brick as defined by the Sprite class are preserved in the array container. This is truly a wonderful thing to behold.

The required code for collision detection between bat and ball as well as ball and multiple bricks follows:

1. Frame action script for multiple bricks

- Add the array container for the bricks
- Create the bricks and load them into the array
- Add the bricks in two rows to the stage
- Comment out or remove unwanted code
- Add the brick array to the game

```
//declare variables
var brickArray:Array = new Array(20);
// attach movie to stage
attachMovie("Ball", "ball", 100);
attachMovie("Bat", "bat", 200);
//attach movie and place bricks on stage
for (var i=0; i<20; i++) {
    attachMovie("Brick", "brick"+i, 300+i);
    //load bricks into array
    brickArray[i] = _root["brick"+i];
    //adjust size of brick to fit stage
    //change width
    brickArray[i].setWidth(Stage.width/10);
    //change height
    brickArray[i].setHeight(15);
    //place bricks on stage screen
    //adjust values dependent on size of brick
    if (i<10) {
        brickArray[i].setX(5+brick0.getWidth()*i);
        brickArray[i].setY(10);
    } else {
        brickArray[i].setX(5+brick0.getWidth()*(19-i));
        brickArray[i].setY(10+brick0.getHeight());
    }
}

/*
attachMovie("Brick", "brick", 300);
//position brick on stage at top left
brick._x=5;
brick._y=10;
*/

//setup
myGame:Game = new Game(ball, bat, brickArray);
```



2. Game controller class

- Add the brick array to the game
- Modify code to collision test each brick in array

```
class Game {
    //===== declare variables
    private var dir:String;
    private var ball:Sprite;
    private var bat:Sprite;
    //private var brick:Sprite;
    private var brickArray:Array;

    //===== constructor
    public function Game (_ball:Sprite, _bat:Sprite, _brickArray:Array) {
        ball=_ball;
        bat=_bat;
        //brick=_brick;
        brickArray=_brickArray;
    }

    //===== public methods
    // move all sprites
    public function moveSprites(_dir:String):Void {
        dir = _dir;
        moveBall();
        moveBat(dir);
    }

    }

    public function checkCollisions():Void {
        if(isCollision(bat,ball)) {
            // do something
            ball.reBoundY();
        }
        // loop to test all bricks
        for (var i=0; i<20; i++) {
            if(isCollision(brickArray[i],ball)) {
                // do something
                ball.reBoundY();
                brickArray[i].reMove();
            }
        }
    }

    // ===== private methods
    // move the ball
    private function moveBall() :Void {
        ball.moveSprite();
    }

    // move the bat in direction
    private function moveBat(_dir:String):Void {
        bat.moveDir(_dir);
    }

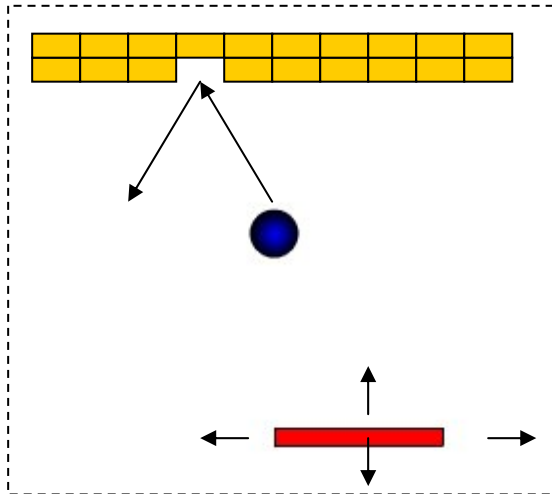
    // hit test two movie clips
    private function isCollision(_mc1:Sprite,_mc2:Sprite):Boolean {
        if(_mc1.hitTest(_mc2))
            return true;
    }
}
```





Checkpoint 6

Test your project now



Functionality achieved:

- All phase five functions
- Multiple bricks (20) in two rows.
- Ball bounces off brick which is then removed
- Continues for each brick

Next phase of development:

- check To do list

Phase 7, 8, 9, ...

How long is a piece of string?

Development from this point is up to the individual in terms of time, expertise and creativity available. The basis for the game's structure has been set. This would be a good point to take students who then have to continue development to include other essential game elements, such as scoring.

To do...

1. Scoring
2. Explosions
3. More bricks in pattern
4. Bonus bricks (different colours)
5. Lives
6. ...

Planning notes

